Interactive Editing Systems: Part I

NORMAN MEYROWITZ AND ANDRIES VAN DAM

Department of Computer Science, Box 1910, Brown University, Providence, Rhode Island 02912

Many daily tasks, whether done with conventional tools or with computers, can be viewed as *editing* tasks: tasks in which the state of some target entity is changed by the user. This article, Part I of a two-part series, examines computer-based *interactive editing systems*, which allow users to change the state of targets such as manuscripts and programs.

This paper is a tutorial that defines terms and introduces issues for the novice, and provides a reference for the more knowledgeable reader The aim is to provide a comprehensive and systematic view of the features of typical systems, highlighting substantive similarities and differences. User and system views of the editing process are provided, a historical perspective is presented, and the functional capabilities of editors are discussed, with emphasis on user-level rather than implementation-level considerations.

Categories and Subject Descriptors: D.2 2 [**Software Engineering**]: Tools and Techniques—*user interfaces*, D.2.3 [**Software Engineering**]: Coding—*prettyprinters*; *program editors*, H.4 1 [**Information Systems Applications**]: Office Automation—*equipment*; *word processing*; I.7.0 [**Text Processing**]: General; I.7.1 [**Text Processing**]: Text Editing—*languages*, *spelling*; I 7.2 [**Text Processing**]. Document Preparation—*format and notation*; *languages*, *photocomposition*; I.7.m [**Text Processing**]: Miscellaneous

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases Syntax-directed editors, structure editors

INTRODUCTION

The interactive editor[1] has become an essential component of any computing environment. It uses the power of the computer for the creation, addition, deletion, and modification of text material such as program statements, manuscript text, and numeric data. The editor allows text to be modified and corrected many orders of magnitude faster and more easily than would manual correction.

Though editors have always been deemed important tools in computing systems, they have only recently become a fashionable topic of research, as they become key components in the office of the future. No longer are editors thought of as tools only for programmers, or for secretaries transcribing from marked-up hard copy generated by authors. It is now increasingly realized that the editor should be considered the primary interface to the computer for all types of *knowledge workers*, as they compose, organize, study, and manipulate computer-based information. Unlike the literature in areas such as programming languages or operating systems (with rich collections of written materials, from basic definitions and tutorials to com-

[1] "Editor" throughout refers to an interactive editing program, not to an author/user correcting a document

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission
© 1982 ACM 0010-4892/82/0900-321 $00.75

CONTENTS

INTRODUCTION
1 GENERAL OVERVIEW
1.1 The Editing Process
1.2 The Editor: A User Viewpoint
1.3 The Editor: A System Viewpoint
2 HISTORICAL DEVELOPMENT OF EDITORS
3 FUNCTIONAL CAPABILITIES
3.1 Traveling
3.2 Viewing
3.3 Editing
3.4 Miscellaneous Capabilities
4 CONCLUSION: PART I

plex formalisms, analyses, and implementation strategies), the literature in the field of editing consists primarily of functional descriptions of particular editors. Little work has been done to standardize terminology or to create a framework for comparing, contrasting, and analyzing editing systems.

This two-part series is designed as a starting point for developing such a framework. Our intended audience is composed of two distinct groups: those with only limited experience with an interactive editor and those with a broader background in the field. Part I is tutorial in nature and is meant for those with only superficial experience in using an editor to create program or document text. It is a reasonably comprehensive introduction to text editing. As such, it is not meant to be read in its entirety by the editor designer, but rather to be used as a reference. It is here that we present our definitions and overview of the field, making explicit what has previously been largely implicit or passed on as "oral tradition." We look at document editing from a user and system standpoint, at some of the major historical developments, and at the functional capabilities of editors.

Part II presents technical details of specific editors, using the terminology and concepts laid out in Part I. The reference list for both parts is included at the end of Part II.

Several topics directly related to interactive editing will **not** be covered here; instead, we supply references to surveys in those fields. In particular, text-formatting techniques are left to Furuta et al. [FURU82], implementation of editors is covered by Rice and van Dam [RICE71] and Finseth [FINS80], editor evaluation is surveyed by Embley and Nagy [EMBL81], and graphics editors are described in Newman and Sproull [NEWM79] and Foley and van Dam [FOLE82].

1. GENERAL OVERVIEW

1.1 The Editing Process

An *interactive editor*[2] is a computer program that allows a user to create and revise a target *document.* We use the term "document" to include targets such as computer programs, text, equations, tables, diagrams, line art, and halftone or color photographs—anything that one might find on a printed page. In this paper we restrict our discussion to *text editors* (in which the primary elements being manipulated are character strings of the target text) used for manuscript and program production, and to *structure editors* (in which the primary elements being manipulated are portions of some generic structure such as a tree).

The document-editing process is an interactive user-computer dialogue (1) to select what part of the target is to be viewed and manipulated, (2) to determine how to format this view on-line and then to display it, (3) to specify and execute operations that modify the target document, and (4) to update the view appropriately.

Selection of the part of the document to be viewed and edited involves first *traveling* through the document to locate the area of interest with operations such as next screenful, bottom, and find pattern. Next, having specified **where** the area of interest is, the selection of **what** is to be viewed and manipulated there is controlled by *filtering.* Filtering extracts the relevant subset of the target document at the point of interest, such as the next screen's worth of text or the next statement. *Formatting* then deter-

[2] In this paper, italic type is used to introduce concepts and terms. Sans serif type is used to set off editor commands. Boldface type is used for emphasis.

mines **how** the result of the filtering will be seen as a visible representation, *the view*, on a display screen or hard-copy device.

Only in the actual *editing* phase is the target document created or altered per a set of operations, commonly including insert, delete, replace, move, and copy. The editing functions are often specialized to operate on *elements* meaningful to the type of editor, such as single characters, words, lines, sentences and paragraphs for manuscript-oriented editors, and keywords and source language statements for program-oriented editors.

In a simple scenario, then, the user might *travel* to the end of a document. A screen's worth of text would be *filtered*, this subset would be *formatted*, and the view would be displayed on an output device. The user then, for example, could delete the first three words of this view.

1.2 The Editor: A User Viewpoint

The user of an interactive editor is presented with a *conceptual model* of the system, which is the designer's abstract framework on which the editor and the "world" in which it operates are based, and with a *user interface*, the collection of tools and techniques with which the user communicates with the editor.

The conceptual model, in essence, provides an easily understood abstraction of the target document and its elements, and a set of guidelines with which to anticipate the effects of operations on these elements. Conceptual models range from those that are hardly visible to the user and not very cohesive or thorough, to those that are well articulated and provide a consistent and complete framework both for using and for implementing the system. In others, the conceptual model is incomplete; it is insufficient to describe more than a very cursory notion of the system. Some of the early *line editors* simulated the world of the keypunch, allowing operations upon numbered sequences of 80-character card image lines, either within a single line or upon an integral number of lines. Some more modern *screen editors* define a world in which a document is represented as a quarter-plane of text lines, unbounded both downward and to the right. The user sees through a cutout only a rectangular subset of this plane at any time on a multiline display screen, but can move the cutout both left and right and up and down to see other portions of a document. Operations manipulate portions of this quarter-plane without regard to line boundaries.

The user interface contains the *input devices*, the *output devices*, and the *interaction language* of the system. Examples of each of these are discussed in the following subsections. Along with the user interface, the user is often given documentation that may include a description of the conceptual model, a high-level description of the system architecture in user-level terminology, a user's guide detailing the syntax and semantics of the interaction language, and a tutorial that provides operational definitions and demonstrates typical situations with examples.

Each individual forms a personal *user model* of an editing system, partly extrapolated from information provided in the recorded documentation or passed on by "experts" and partly based on repeated use of the system. The user model may differ from the conceptual model in several ways. First, it can be thought of as a subset of the conceptual model. For example, a user experienced in document preparation may have no notion of the language-specific commands for correct program indentation; similarly, a programmer may have no notion about the features provided for on-line table manipulation. Second, when the user consistently uses combinations of primitives to perform common operations in ways not originally encompassed by the conceptual model, the user model can be thought of as an extension of the conceptual model. Third, the user model can be thought of as an operationally equivalent but logically different form of the conceptual model. For example, rather than considering his or her actions to be moving a cutout over various parts of a document as described above, the user may consider the cutout as stationary and the document as scrollable horizontally and vertically past this cutout.

The user model is a personalized, high-level understanding of the conceptual

model provided, of the manipulable entities and their interrelationships, of the set of operations allowed on the entities, and of the interaction language used to invoke these operations. The user model is not simply descriptive but prescriptive as well: to perform editing tasks, the user employs the user interface based on his or her user model.

1.2.1 User Interface

1.2.1.1 Input Devices. Input devices are used for three main purposes: (1) to enter elements, (2) to enter commands, and (3) to designate editable elements. These devices, as used with editors, can be divided into three categories: *text* devices, *button* devices, and *locator* devices [GSPC79, FOLE82].

Text or *string* devices are typically typewriterlike keyboards on which a user presses and releases a key to send to the CPU or to an I/O controller a unique code for each key. Essentially all current computer keyboards are of the QWERTY variety. This variation of keyboard, named for the first six letters in the second row of the keyboard, was invented by Christopher Latham Sholes in the 1860s. The strange key layout was done for purely mechanical reasons—letters most commonly used together were placed as far apart as possible so that mechanical typewriter keys would not clash. As surveyed in MONT82, several experimental text input devices have been constructed. The Dvorak Simplified Keyboard rearranges the keys to reduce fatigue and increase typing speed; despite experimental evidence showing its superiority, it has not caught on strongly. The Montgomery "wipe-activated" keyboard requires the user not to press keys, but rather to "wipe" a stylus across the keyboard surface. Letters comprising common digrams and trigrams, such as TH and ING, are placed next to each other on the keyboard; the user simply wipes the wand from left to right across the letters, rather than typing two or three separate keystrokes. NLS's keyset [ENGE68] consists of a pad of five long keys, similar in shape to white piano keys, and provides a method for entering small text additions or corrections. Each key corresponds to a position in a 5-bit binary word called a "chord"; by depressing the proper combination, the user can represent $2^5 - 1 = 31$ numbers or ASCII codes (binary "00000" is not counted, as this means no keys are being pressed).

An alternative to direct keyboard input is *optical character recognition* (*OCR*). Typically a standard electric typewriter, (inexpensively) equipped with an OCR type element, is used for typing the text on normal paper. This paper is then fed through an optical character reader, an electromechanical device that scans the page and translates each character into the proper digital representation for that computer system. This technique allows the continued use of preexisting facilities: the typewriters become off-line "links" to the computer facilities. Although many OCR systems also allow rudimentary editing of the raw text stream via *character-* and *line-delete* control characters inserted in the text, this cannot compete with the convenience and essentially instant editing facility of on-line input. While OCR is often seen as an inexpensive way to begin "computerization," it is contrary to the spirit of online authoring, in which the author is able to express his or her thoughts and experiment with a composition from its inception.

Button or *choice* devices generate an interrupt or set a system flag, usually causing invocation of an associated application program action. They typically are grouped as a set of special function keys on the alphanumeric keyboard or on the display itself. Alternatively, buttons are often simulated in software by having the user choose text strings or symbols displayed on a screen.

Locator devices are *x-y* analog-to-digital transducers that position a cursor symbol on the screen by continually sampling the analog values produced by the user's movement of the device. They include *joysticks*, *trackballs*, *touch screen panels*, *data tablets*, and *mice*. The latter two are the most common locator devices for editing applications. The data tablet is a flat, rectangular, electromagnetically sensitive panel. Either a ballpoint-pen-like *stylus* or a *puck*, a small (approximately 3 inch square × 1 inch high) device that fits in the palm of the hand, is moved over the tablet surface.

The tablet returns to a system program the coordinates of the point on the data tablet on which the puck or stylus is currently located. The program can then map these data tablet coordinates to screen coordinates and move the cursor to the corresponding screen position. The mouse is another hand-held device about the same size as the puck. As it is moved on a flat surface, the motion of the mouse causes relative changes in x and y to be sampled by a system program. These mouse coordinates are again mapped to screen coordinates to move a cursor on the screen.

A locator device coupled with a button device allows the user to specify either a particular point on the screen at which text should be inserted or deleted, or the start and endpoints of a string of characters to be operated upon. In fact, the mouse and puck usually have built-in buttons for the user to signal a selection. When the cursor has been positioned over an element, the user presses a button to indicate the selection; the system correlates the cursor position with the element that it covers and performs the appropriate action.

Text devices with arrow (cursor) keys are often used to simulate locator devices. These keys each show a pictured arrow, pointing up, down, left, or right, respectively. Pressing an arrow key typically generates an appropriate character sequence, which the program then translates to update the cursor position in the direction of the arrow on the key pressed. *Up*, *down*, *left*, and *right* keys must be used sequentially to position the cursor a character at a time, aided typically by continuous movement in "repeat mode," which the device enters if a key is held down for more than, say, a second.

Still in the research stage, *voice input devices*, which translate spoken words—both literal text and commands—to their textual equivalents, may prove to be the text devices of the future. While currently restricted to a small vocabulary (typically fewer than 1000 words), voice recognizers may soon be commercially viable for command recognition.

1.2.1.2 Output Devices. Formerly limited in range, output devices for editing are diversifying. The output device serves to let the user view the elements being edited and the results of the editing operations. The first-generation output devices were the (now largely obsolete) teletypewriters and other character-printing terminals, which generated output on paper. Next, "glass teletypes" based on *cathode ray tube* (*CRT*) technology used the CRT screen essentially to simulate a hard-copy teletypewriter, although a few operations, such as backspace, were performed more elegantly. Today's advanced CRT terminals use hardware assistance for such features as moving the cursor, inserting and deleting characters and lines, and scrolling lines and pages. The new generation of *professional work stations*, based on personal computers with high-resolution raster displays, supports multiple proportionally spaced character fonts to produce realistic facsimiles of hard-copy documents. Thus the user can see the document portrayed essentially as it would look when printed on paper.

While many editors, especially traditional ones, have interfaces that allow them to run on both CRT and hard-copy terminals, we concentrate on the more sophisticated capabilities that can readily be implemented only on CRT terminals with user-manipulable cursors. Much of the discussion does, however, apply to both hard-copy and CRT output devices.

1.2.1.3 Interaction Language. The interaction language of a text editor can be divided into three parts: the *semantic component*, the *syntactic component*, and the *lexical component* [FOLE82]. The semantic component of the language specifies functionality: what operations are valid for each element, what information is needed for the manipulation of each element, what the results of the operations are, and what errors may occur. The semantic component defines meanings of the specified operations, not particular data structures or dialogues to implement those operations.

The syntactic component specifies the input and output rules by which *tokens*, as the atomic elements, are put together to form sentences in the *grammar* of the language. In terms of editor input, the set of tokens might include character strings, commands, and screen positions. In terms of output, the set of tokens might include

character strings, lines, and formatted paragraphs. The syntax must be easy for the user to learn and remember, and must follow naturally from the conceptual model of the system. The syntax of interaction languages is generally one of three sorts: *prefix*, *postfix*, or *infix*. A prefix (verb/noun) command specifies the operation desired, followed by the element(s) which that operation will affect. A postfix (noun/verb) command specifies just the opposite: the element to be operated upon is specified first, followed by the operation. An infix (noun/verb/noun) command is a cross between the two former types: the operation is surrounded by its operands. Typically infix only is needed when more than one operand exists. Many languages are basically postfix, but rely on infix in the cases where more than one operand is needed. In a *cross-product* interface, the user can match a noun from a list of nouns with a verb from a list of verbs to form the appropriate command. In other types of interfaces, the verb/noun distinction is not always clear-cut; a single editor command, such as delete-word, can be composed of both a verb and noun. Here the user specifies the command delete, and the element is particularized to the nearest unit of the corresponding type word.

The lexical component specifies the way in which *lexemes*, information from the input devices or for the output devices, are combined to form the tokens used by the syntactic component. Thus typing the lexemes D, -, W, followed by a carriage return, would result in the delete-word token.

The interaction language is generally one of several common types, based on the manner of specifying lexemes. The *typing-* or *text command-oriented* interface is the oldest of the major editor interfaces. Here, the user communicates with the editor by typing text strings both for command names and operands. These strings are sent to the editor and are usually echoed on the output device.

Typed specification often requires the user to remember the exact form of all commands, or at least of their abbreviations. (Some systems will prompt the user with valid choices if an ambiguous command abbreviation is typed.) If the interaction syntax is complex, the user must continually refer to a manual or an on-line "help" command for a description of less frequently used commands. Additionally, typing is time consuming, especially for the beginner. The *function-key* interface addresses these deficiencies. Here each command has associated with it a marked key on the user's keyboard. For example, the insert character command might have associated with it a key marked IC. Forgetting the commands is unlikely, since they are literally at the user's fingertips. For the common functions, usually only a single key need be pressed. Function-key syntax is typically coupled with cursor-key movement for specifying operands, thereby eliminating much typing. Advocates cite "muscle memory" of key location as a prime advantage of function key interfaces.

For less frequently invoked commands or options in a function-key editor, an optional textual syntax may be used. More commonly, special keys "shift" the standard function key interpretations, just as the shift key on a typewriter shifts the standard interpretation of a key from being lowercase to being uppercase. As an alternative to shifting function keys, the alphanumeric keyboard is often *overloaded* to simulate function keys. The user again uses special shift keys—the most common being the **control**[3] key (CTRL) depressed simultaneously with normal keys to generate new characters that are interpreted like function keys. Generally, functions are assigned to alphanumeric keys in two ways: topologically and mnemonically. In topological layout, functionally similar keys are grouped in close proximity; for example, in Brown's *bb* [REIS81], the control key coupled with the Q, W, E, or R keys correspond to delete word, delete to end of line, delete to blank line, and delete paragraph, respectively, while the keys directly below, A, S, D, F, coupled with the control key, correspond to insert word, insert to end of line, insert to blank line, and insert paragraph. Another topological layout is seen in Wordstar [MICR81], in which the cursor commands, up, down, left, and right, are bound to the

[3] In examples, the control key will be abbreviated CTRL- and the escape key ESC

CTRL-E, CTRL-X, CTRL-A, and CTRL-F keys to make use of their relative directions on the keyboard. Mnemonic layout binds commands to keys whose letters or symbols invoke some type of mental image or connection. For instance, in the EMACS [STAL80] default key bindings, CTRL-B, CTRL-E, CTRL-F, CTRL-R, and CTRL-K stand for Backward character, End of line, Forward character, Reverse search, and Kill line, respectively.

Typing-oriented systems require familiarity with the system and language, as well as some expertise in typing. Function-key-oriented systems often have either too few keys, thus requiring keyboard-overloading by binding single keys to several interpretations and necessitating multiple keystroke commands, or have too many unique keys, resulting in an unwieldy keyboard. In either case, even more agility is demanded of the user than by a standard keyboard. The *menu-oriented* user interface is an attempt to address these problems. A *menu* is a multiple-choice set of text strings or *icons* (graphical symbols which represent objects or operations) from which the user can select items to perform actions [GOLA79]. The editor prompts the user with a menu of only those actions that may be taken at the current state of the system. The user knows that if a command appears in the menu, it can be selected, and the typical "you can't do that operation in this (*x*) mode" error messages that occur with other types of interfaces are eliminated.

One problem with a menu-oriented system can arise when there are many possible actions and multiple choices required to complete an action. Since the display area for the menu is usually rather limited, the user might be presented with several consecutive menus in a hierarchy before the appropriate command and its options appear. Since this can be annoying and detrimental to the efficiency of a seasoned user, some menu-oriented systems allow the user to turn off menu control, leaving a language- or function-key-oriented editor as a base. Others have the most-used functions on a main command menu and have secondary menus to handle the less frequently used functions. Still others display the menu only when the user specifically asks for it. For instance, in more modern editors based on a Smalltalk-80[4]-like interface [TESL81, GOLA82] (see Figure 1), *pop-up* menus, in response to the user's button push, instantly appear on some part of the screen near the cursor (perhaps temporarily overlapping some existing information) to give the user the full choice of applicable commands. The user selects the appropriate command with a mouse; the system executes the command and the menu disappears instantly. Interfaces like this, in which prompting and menu information are given to the user at little added cost and little degradation in response time, are becoming increasingly popular.

While the semantic component of the language is similar from system to system (at least for simple operations), the lexical and syntactic specification of commands vary widely. For example, to delete the word "bazinga" in an editor using typed commands from a keyboard without cursor keys, one might search for an occurrence of the pattern bazinga and then type delete/bazinga; in another, using typed commands with cursor keys, one might type dw after driving a cursor to point at any character in the word bazinga; in a function-key driven system, one might press the del word key (or keys) after pointing at bazinga; and in a menu-oriented system, one might point to and then select the b in bazinga, adjust the selection to the entire word by pushing the adjust button, and then select the delete icon from the displayed menu.

The interaction language can be viewed as a layered architecture: similar semantics can be reflected in a variety of syntaxes, while similar syntaxes can be reflected in a variety of lexical styles. For example, given semantic routines that perform insertion and deletion in an unbounded stream of characters, designers could provide a prefix or postfix syntax for specifying insert or delete commands. Similarly, given either the prefix or postfix syntax, designers could provide typing-oriented, function-key-oriented, or menu-oriented lexical input styles. This layering invites device independence,

[4] Smalltalk-80 is a registered trademark of Xerox Corporation.

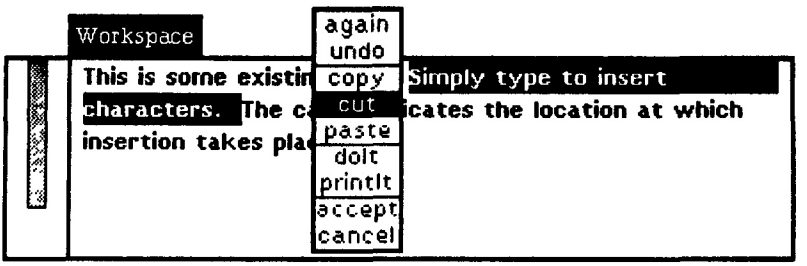

Figure 1. A Smalltalk-80 pop-up menu. Here the arrow cursor, driven by the mouse, points to the cut item in the pop-up menu temporarily overlapping the text window. When the proper mouse button is pressed, the command is executed and the menu, no longer needed, disappears.

since differing lexical forms can be mapped into the same syntax. The layering also allows syntax independence, since differing syntactical styles can be mapped into the same semantic operations.

1.3 The Editor: A System Viewpoint

Editors in general follow an architecture similar to that in Figure 2, regardless of the particular computers on which they are implemented and the features they offer.

The *command language processor* accepts input from the user's input devices, lexically analyzes and tokenizes the input stream, syntactically analyzes the accumulated stream of tokens, and, upon finding a legal composition of tokens, invokes the appropriate semantic routines.

At the syntactic level, the command language processor, like a programming language processor, may generate an intermediate representation of the desired editing operations instead of explicitly invoking the semantic routines. This intermediate representation is decoded by an interpreter that invokes the proper semantic routines. This allows the use of multiple interaction syntaxes with a single set of semantic routines that are driven from a common intermediate representation.

The semantic routines invoke traveling, editing, viewing, and display. While editing operations are always specified explicitly by the user, and the display operations implicitly by the other three categories of operations, traveling and viewing operations may be either explicitly specified or implicitly invoked by the editing operations. In fact, the relationship between these classes of operations may be considerably more complicated than the simple model of Section 1.1 (travel to determine **where** the selection should take place, filter to select **what** is to be viewed and manipulated, format to determine **how** the view is to appear, then edit and reformat) might suggest. In particular, there need not be a simple one-to-one relationship between what is "in view," that is, what is displayed on the screen currently, and what can be edited. To illustrate this, we take a closer look below at the components of Figure 2 which are meant to be conceptual entities.

In editing a document, the start of the area to be edited is determined by the *current editing pointer* maintained by the *editing component* of the editor, the collection of modules dealing with editing tasks. The current editing pointer can be set or reset explicitly by the user with a traveling command such as next paragraph and next screen, or implicitly by the system as a side effect of the previous editing operation, such as delete paragraph. (The *traveling component* of the editor actually performs the setting of the current editing and viewing pointers, and thus the point at which the viewing and/or editing filtering begins.) When the user issues an editing command, the editing component invokes the *editing filter*. The editing filter filters the document to generate a new *editing buffer* based on the current editing pointer as well as the editing filter parameters. These parameters are specified both by the user and the system, and provide such information as the range of text that can be affected by the operation, for example, the "current line"

in a line editor or the "current screen" in a display editor. Filtering in both the case of editing and viewing may be defaulted to the selection of contiguous characters at the current point or may depend on more complex user specifications pertaining to the content and structure of the document to gather not necessarily contiguous portions of the document, as discussed in Section 3.1. The semantic routines of the editing component then operate on the editing buffer, which is essentially a filtered subset of the document data structure. (Note that this explanation is at the conceptual level—in a given editor, filtering and editing may be interleaved, and no explicit editing buffer created.)

Similarly, in viewing a document, the start of the area to be viewed is determined by the *current viewing pointer* maintained by the *viewing component* of the editor, the collection of modules responsible for determining the next view. The current viewing pointer can be set or reset explicitly by the user with a traveling command or implicitly by the system as a side effect of the previous editing operation. When the display needs to be updated, the viewing component invokes the *viewing filter*. The viewing filter filters the document to generate a new *viewing buffer* based on the current viewing pointer as well as the viewing filter parameters. These parameters, again, are specified both by the user and the system, and provide such information as the number of characters needed to fill the display and how to select them from the document. The viewing buffer may contain the "current line" or the null string (for "silent" feedback) in line editors, while in screen editors it may contain a rectangular cutout of the quarter plane of text. This viewing buffer is then passed to the *display component* of the editor, which maps it to a *window* or *viewport*, a rectangular subset of the screen, to produce a display.[5]

The editing and viewing buffers, while independent, can be related in many ways. In the simple case they are **identical**, as in the case of screen editors, in which the user edits the material directly in view on the screen, rather than specify material with typed commands (see Figure 3).

The editing and viewing buffer can also be **disjoint**. For example, in the Berkeley UNIX[6] editor *ex* [JOY80a], a user might travel to line 75, and after viewing it, decide to change all occurrences of "ugly duckling" to "swan" in lines 1 through 50 of the file by using the substitute command:

```
1,50s/ugly duckling/swan/g
```

As part of this editing command, there is implicit travel to the first line of the file, lines 1 through 50 are filtered from the document to become the editing buffer, and successive substitutions take place in the editing buffer without corresponding updates of the view. If the pattern is found, the current pointers are moved to the last line on which it was found, and that line becomes the default contents of both the editing and viewing buffers, while if the pattern is not found, line 75 remains as the default editing and viewing buffers.

The editing and viewing buffers can be **partially overlapping** on a screen when the user specifies a search to the end-of-document starting at a character position in the middle of the screen. Here the editing filter creates an editing buffer that contains the document from the selected character to the end of the document, while the viewing buffer contains the part of the document that is visible on the screen (only the last part of which will be in the editing buffer).

Finally, the editing and viewing buffers may be **properly contained** in one another. As an example of the editing buffer contained in a larger viewing buffer, the CPT word processing editor [SEYP79] al-

[5] The term "viewport" is standard for "visible representation on the screen" in computer graphics terminology, while in editing terminology, the term "window" is used loosely to mean both the viewing buffer and the mapped representation of the viewing buffer on the screen. Unfortunately, the term "window" in graphics terminology is analogous to our "viewing buffer," lending more confusion to the definitions We have tried to limit the use of the term "window" to reduce confusion, but when we do, we use it in the editing sense of the "mapped representation of the viewing buffer on the screen." We use the term "view" to mean a "display of a filtered subset of a document in a window."

[6] UNIX is a trademark of Bell Laboratories.

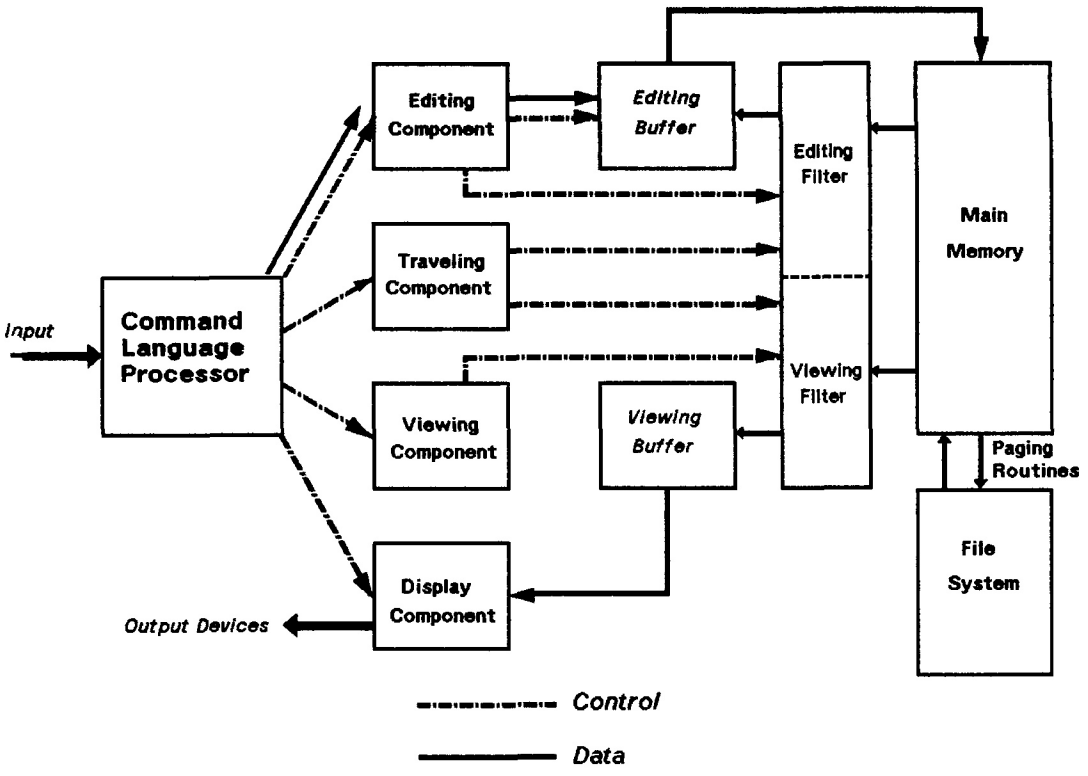

Figure 2. The editor· a system architecture.

lows full 8½ × 11-inch page views. The user must scroll a page to put a line to be edited at the *typing bar*, analogous to rolling the platen of a typewriter to align the desired line with the typing element. Here the editing buffer corresponds to the one-line subset of the document which is in the middle of the viewing buffer. Conversely, a number of traditional editors, limited by low-speed, line-oriented output devices, provide a large editing buffer. The viewing buffer is a small initial subset of the editing buffer to allow quick regeneration of the view; the entire scope of the editing operation need not be in view. The FRESS editor [VAND71a, PRUS79] allows the user to select the number of lines and the number of characters per line for the viewing buffer (the default being a single line the width of the display device for alphanumeric terminals) and also allows the user to select the size of the editing buffer (the default being 2000 characters); the viewing filter selects as many characters as necessary, starting at the beginning of the editing buffer, to provide the viewing buffer for the necessary display.

Windows typically cover either the entire screen or a rectangular portion of it. Mapping viewing buffers to windows that cover only part of the screen is especially useful for editors on modern, high-resolution raster-graphics-based work stations, as they allow the user to examine and to interact with multiple views, for inter- and intrafile editing and "cutting and pasting." The notion of multiple viewing buffers and multiple windows showing differing portions of the same or multiple files at the same time on the screen is designed into such editors. Alternatively, on graphics-based work stations with multiwindow *display managers* or *window managers* that allow the user to manipulate the placement of windows on a screen [LRG76, TEIW77, LANT79, MEYR81, SYMB81, TESL81], the user can run a one-viewing-buffer editor in each of the many windows. It is through the support of the display manager, **not** of the editor, that the

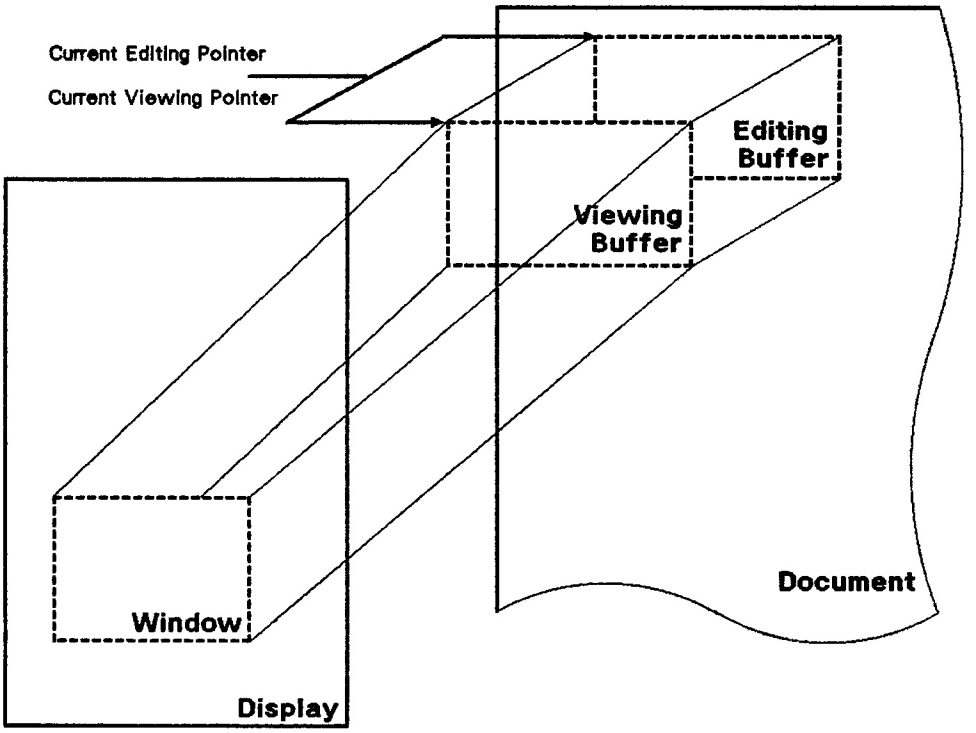

Figure 3. Elements of the editing component. The diagram above shows the relationship of the current operating pointers in a document file to the editing buffer and the viewing buffer. In this diagram, the file is indicated as a quarter-plane of text. The current editing pointer points to the start of the editing buffer. The current viewing pointer points to the start of the viewing buffer. The two point to the same place in this exploded diagram since the viewing buffer coincides with the editing buffer in this example.

user can cut and paste text between multiple windows containing portions of the same file or portions of multiple files.

As a comment on slow progress in the editing field, we note that in the database and graphics fields, the notion of multiple simultaneous views of the same or related data is well understood and commonly accepted. Although the facility for multiple views of documents at varying levels of detail and appearance has been available in the NLS editor since the mid-1960s [ENGE68] (see Section 2 and Part II, Section 1.5), this powerful organizational mechanism is only now beginning to gain acceptance in the editing community.

The viewing buffer-to-window mapping is accomplished by two components of the system. First, the viewing component formats an ideal view, often expressed in a device-independent intermediate representation. This ideal view can range from simply a window's worth of text, arranged so that lines are not broken in the middle of words ("ragged right" justification), to a facsimile of a page of fully formatted/typeset text with equations, tables, and figures.

Second, the *display component* takes this idealized view from the viewing component and simply maps it to a physical output device in the most efficient manner possible. If a view that has been computed by the viewing component cannot be fully displayed because the window is partly obscured by another window, this mapping becomes somewhat more complicated.

Updating of a full-screen display connected over low-speed lines (1200 baud or less) is slow if every modification requires a full rewrite of the display surface. Much research is concerned with optimal screen-updating algorithms that compare the current version of the screen with the following

version and, using the innate capabilities of the terminal, write only those characters needed to generate a correct display. Typical algorithms for this intelligent screen update are described in ARNO80, BARA81, GOSL81, and STAL81.

Device-independent output, like device-independent input, promotes interaction language portability. This decoupling of editing and viewing operations from display functions for output is important for purposes of portability: it is unwieldy to have a different version of the editor for every particular output device. Many editors make use of a *terminal control database* [JOY81]. Instead of having explicit terminal-control sequences in display routines, these editors simply call terminal-independent library routines, such as scroll down or read cursor position, that look up the appropriate control sequences for the host terminal. Consequently, adding a new terminal merely entails adding a database description of that terminal.

In addition to the interrelated traveling, viewing, editing, and displaying components, special *utility components*, such as spelling checkers/correctors and word-count analyzers, may exist to provide a variety of services to aid the user in document production.

The components "communicate" with a user document on two levels: in main memory and in the disk file system. At the beginning of an editing session the editor first asks the file system to open the appropriate file and then reads the entire file or files, or portions thereof, into main memory. Loading an entire document into main memory may be infeasible. Yet if only a portion of the document were resident in main memory and if many user-specified operations entailed a disk-read by the editor to load the affected portion, editing would be unacceptably slow. In many modern systems this problem is solved by mapping the entire file into virtual memory and letting the operating system perform efficient demand paging. Alternatively, in systems without virtual memory or with limited virtual memory per user, *editor paging routines* are required. These read in one or more logical chunks of a document, called pages (although there is typically no correspondence between these pages and hard-copy document pages or virtual memory pages), into main memory, where they reside until a user operation requires another piece of the document. In either case, documents are often represented not as sequential strings of characters, but in an *editor data structure* that allows addition, deletion, and modification with a minimum of I/O and character movement. When stored on disk, the file may be stored in terms of this data structure or in an editor-independent, general-purpose format (e.g., as character strings with embedded control characters such as linefeed and tab).

1.3.1 Configurations

Editors function in the three basic types of computing environments: *timesharing*, *stand-alone*, and *distributed*. Each type of environment imposes some constraints on the design of an editor. The timesharing editor must function swiftly within the context of the load on the computer's processor, primary memory, and secondary memory. The editor on a stand-alone system must have access to the functions that the timesharing editor obtains from its host operating system; these may be provided in part by a small local operating system or may be built into the editor itself if the stand-alone system is dedicated to editing. The editor operating in a distributed resource-sharing local network must, like a stand-alone editor, run independently on each user's machine, and must, like a timesharing editor, contend for shared resources such as files.

Some timesharing-based editing systems take advantage of local (terminal-based) hardware to perform editing tasks. These *intelligent terminals* have their own microprocessors and local buffer memories in which editing manipulations can be done, thus saving the time needed to read and write main computer memory, but adding tricky data structure synchronization problems. Small actions are not controlled by the CPU of the host processor but are handled by the local terminal itself. In a system using an IBM 3270 series terminal, for example, the editor sends a full screen of material from the mainframe computer to

the terminal. The user is free to add and delete characters and lines; when the buffer has been edited, its updated contents are transmitted back to the mainframe. The advantage of this scheme, that the host need not be concerned with each minor change or keystroke, is also the major disadvantage. With a nonintelligent terminal the CPU "sees" every character as it is typed in and can react immediately to perform error checking, to prompt, to update the data structure and to record or "journal" the keystrokes for undoing editing operations (see Section 3.4). With an intelligent terminal, the lack of constant CPU intervention often means that the functionality provided to the user is more limited. Also, local work on the intelligent terminal is lost in the event of a system crash. Conversely, systems that allow each character to interrupt the CPU may not use the full hardware editing capabilities of the terminal because the CPU needs to see every keystroke and provide character-by-character feedback.

As the local-area network of largely self-sufficient work stations becomes the dominant architecture, we may expect that the problem of synchronizing intelligent terminals with timeshared hosts will disappear and that character-by-character feedback will become the norm.

2. HISTORICAL DEVELOPMENT OF EDITORS

The history of editing is one of many complementary development efforts proceeding in parallel. Editors are so numerous and their relationships so cloudy, that it is very difficult to provide an accurate chronology. Rather, we briefly overview some of the important concepts, citing familiar and representative examples. (For a survey of specific editors through 1971, see VAND71b.)

Noninteractive computerized editing began with the manipulation of "unit record" punched cards. The basic unit of information was the 80-column line; the user made corrections on a line-by-line basis, retyping mistyped cards. Compared to toggling in bits at the system console, the card gave the programmer new freedom. One could store information in both human- and machine-readable form, and one could "travel" through this information, changing its order, discovering and fixing errors, recognizing color-coded groups of cards, or simply browsing through the deck.

Punched card decks had many disadvantages, such as the "rearrangement" that resulted when a box was accidentally dropped. More seriously, editing a small part of a large document required feeding the entire document, often thousands of cards, into the reader for every change. To correct small errors such as single-character errors or double-character transpositions, the user had to retype the offending characters and to replicate the other characters with the duplication facilities of the keypunch. Replacing a word with a word of a different size necessitated duplicating all characters prior to the word and retyping all the characters from the new word forward. If the incorrect card was almost completely filled with characters, inserting a new word might result in overflow of the contents, causing the insertion of one or more new cards to handle the overflow. Performing a global change required manually finding every occurrence of the old pattern and again manually replacing it with the new pattern; if the new pattern were larger than the old pattern, multiple overflows could easily occur.

To address the problems of the punched card in the still predominantly batch environments of the 1960s, *card* or *batch editors* were created. Here the programmer's initial deck of cards was stored as a card-image tape or disk file. Each card was referenced by a unique sequence number. Changes were made by creating an *edit deck* composed of cards containing editing requests, and running the deck through the batch editor program. For example, the request "in card 35, correctly spell the word 'rate'" would be made by simply typing the desired sequence number 35 on one card followed by a card containing the new contents of line 35, or more simply, by composing a card with a sequence number and an editing command, as in

```
35 CHANGE/RATA/RATE/
```

Batch editors removed the problems of dropped cards and of retyping (in many

cases), and, in some versions, provided new operations such as global replacement of a pattern. There were several disadvantages, however. Programmers needed to have a line-printer listing of the entire card deck before making any change. Also, some of the organizational characteristics offered by cards, such as the easy visual inspection of a properly sequenced, color-coded, and well-labeled card box, were lost.

Systems like IBM's MTST [IBM67], which used a Selectric typewriter as an input device and small magnetic tapes and/or cards as storage media, were the forerunners of today's word-processing systems. Because these primitive interactive editors relied on sequential storage media such as magnetic tape or magnetic cards, the user could only step through card images linearly, stopping at lines which needed correction and retyping them. To go backward, the user had to "rewind" and then "fast-forward" the file to the desired place. In addition, the utility of these initial line editors was limited by the typewriters, which supported the viewing of only one line at a time and had very slow printing speeds.

With the advent of timesharing in the mid-1960s, interactive *line editors* were designed that allowed the user to create and modify disk files from terminals. These editors attached either fixed or varying ("varying" meaning "sequential relative to the top of the file") line numbers to lines of limited length (initially 80 characters), allowing the user to reference a unit of information. Early examples of these include ATS [IBM70] and VIPcom [VIP69]. Simple command languages allowed the user to make corrections within a line or even within a group of contiguous lines, using much the same syntax as did batch editors. Some early timesharing editors still restricted the user to forward access; later, this restriction was lifted and the user could scroll both forward and backward through a file. Typically, line editors with bounded-length lines shared the unfortunate property of *truncation*: if an insert or change of characters forced the line to exceed the maximum length, characters were dropped off the end of the line as needed. This implementation "feature," stemming from a conceptual model of the editing process based on simulating punched cards and line printer listings, was only marginally acceptable for program editing and unacceptable for serious manuscript preparation. (In the latter case, automatic creation of a new line upon overflow would have been a trivial fix.)

Another advance was the creation of the *context-driven line editor*, which allowed the user to identify the line containing the target of an operation by specifying a character context pattern for the editor to match, rather than by giving an explicit line number. One of the first examples of the context-driven line editor was the trend-setting editor running on the IBM 7090 as part of Project MAC's CTSS [CRIS65]. Other classic examples include IBM's CMS editor [IBM69] (see Part II, Section 1) and Stanford's WYLBUR [FAJM73]. At this point in the history of editing, users were still forced to think about multiline entities, such as paragraphs and program blocks, as groups of integral lines, usually in card image format; no interline commands were available that would, for example, delete text spanning from the middle of one line to the middle of the next line.

The first break from the 80-column card image came in the form of *variable-length line editors*, typified by Com-Share's Quick Editor (QED) [DEUT67, COMS67]. The main element of operation was still the line, but now each line could be of "arbitrary" length. Initially, these lines were actually limited to some maximum. QED popularized the notion of a "superline" (limited to 500 characters in length), which the on-line display process broke into viewable lines of 80 characters each until the superline was exhausted. While editing across superline boundaries was still impossible in these editors, the probability that a full phrase or sentence that needed editing would fall within a single superline was much greater than with an 80-character limit. Later, true variable-length line editors removed the restriction. By removing the card image orientation of the editor, the variable-length line editor had strong and beneficial impact on the versatility of text processing. Another far-reaching result of the invention of variable-length line editors was that displayed text was no longer considered to be

a one-to-one mapping of the internal representation, but rather a tailored, more abstract view of the editable elements.

Even with superline editors, three basic problems in manuscript editing remained: truncation when the line length was exceeded, inability to edit a string crossing line boundaries, and inability to search for a pattern crossing line boundaries. This last problem is an especially difficult one when transcribing editing changes from formatted hard copy in that even a short phrase that appears on one line within the paper may be spread across two lines in the document's source file, unbeknown to the user. Consider, for example, the familiar "the the" typo problem. If a document being edited on a line editor contained the lines

```
. . . The power of the
the stream editor . . .
```

a search command

```
locate/the the
```

would not find the pattern "the the," since it appears on two separate lines. The *stream editor* concept solved all three problems by eliminating line boundaries altogether: the entire text was considered a single stream or string that was broken into screen lines by display routines. An arbitrary string between any two characters could be defined for searching and editing. HES [CARM69], and FRESS are examples of stream editors. Although the TECO [BBN73] stream editor left the carriage returns and line feeds in the text stream, these could be edited like any other character; they did not serve to isolate one line from the next in editing. In fact, the generality of the TECO stream model has made possible the construction of several sophisticated editors using TECO as a nucleus (see Part II, Section 1).

Another way of dealing with the limitations of line/superline editors was to use the power of multiline display screens, which provided cursor addressability and (possibly) local buffers, to create what are called synonymously *full-screen*, *display*, or *cursor editors*. These editors work either with variable-length lines or with streams, offering the user an entire screenful of text to view and edit without regard to line boundaries. An early example of a time-shared display editor is Stanford University's TVEDIT [TOLL65, MCCA67]. Commands, represented by control character sequences, could be interspersed with the input of "normal" text. Users were able to move the cursor to point to the text they wished to manipulate rather than having to describe text arguments in some awkward syntax. Characters could be replaced by simply typing over them. Characters could be deleted by placing the cursor on the character and pressing the delete control character; characters to the right of the cursor moved left so that the cursor seemed to "swallow" characters. Similarly, for insertions, the characters to the right of the pointer moved to the right, "making room" for the new characters. The TVEDIT concepts and similar work by Ned Irons [IRON72] form the basis of many screen editors in use today.

A major new way of thinking about editing was introduced as early as 1959 by Douglas Engelbart at Stanford Research Institute. His NLS (oNLine System), implemented in the 1960s to create an environment for on-line thinking and authoring, showed the power of display terminals, multicontext viewing, flexible file viewing, and a consistent user interface [ENGE63, ENGE68, ENGE73]. One of the NLS project's many important contributions was the mouse, an input device that is only now being integrated into commercial products. NLS was the first structure editor in that it provided support for text structure and hierarchy, not just for manipulating raw strings of text: the user could manipulate documents in terms of their structure, not only their content.

NLS and other related systems, such as HES, FRESS, and Xanadu[7] [NELS74, NELS81], are particularly important because they view the editor as an **author's tool**, an interactive means for organizing and browsing through information, rather than simply as a mundane tool for altering characters in a single file.

Hansen's EMILY [HANS71] extended the concept of the structure editor and devel-

[7] Xanadu is a registered trademark.

oped the *syntax-directed editor*, in which the structure imposed on a program being edited was the structure of the programming language itself. Users were able to manipulate logical constructs, such as do–while loops and their nested contents, as single units.

In the late 1960s, general-purpose timesharing facilities typically supported only simple interactive line-editing and batch-formatting facilities for line-printer output. These "value-added" facilities were barely adequate to create and modify programs and rudimentary documentation. By the early 1970s, text processing had become sufficiently important to be the single dedicated application on both stand-alone and timeshared ("shared-logic") minicomputers. Since these minicomputers did not need to support general-purpose computing facilities, manufacturers were able to offer comprehensive editing and formatting/typesetting capabilities as well as features oriented toward document production such as database management, information retrieval, work-flow management, and print- and job-queue management that were usually unavailable on general-purpose systems. For a time, owners of these systems often had more text-processing power than those with much more expensive and much larger general-purpose computers. Examples of dedicated word processing systems include CPT, Lanier, DEC Word/11, NBI, and Wang.

An important milestone in text editing and text processing was the early 1970s development and mid 1970s acceptance of the UNIX timesharing system, the first general-purpose computing environment in which text utilities were given as much weight as programming utilities. In UNIX, a suite of utilities (the *ed* text editor, the *troff* text formatter, the *eqn* equation formatter, the *tbl* table formatter, the *refer* bibliographic database and formatter, the *spell* spelling corrector, and the *style* and *diction* text analyzers [KERN82]) introduced and popularized an extensive set of text tools in the general-purpose computing community. At the same time the publishing industries—newspapers, magazines, wire services, the graphics arts—converted wholesale to electronic typesetting and layout of pages, borrowing ideas from traditional computer-based text processing, and also channeling ideas in typesetting and page layout back to the computing community.

In the then-separate area of computer graphics, picture editors were being designed to allow the user to manipulate graphical elements. Interactive drawing techniques from Sutherland's pioneering Sketchpad system [SUTH63] were later incorporated into editor interfaces. The Carnegie-Mellon tablet editor [COLE69] is an example of the use of this technology. In this experimental editor, hand-drawn proofreader's symbols were used to edit displayed text. The symbols were drawn on a data tablet and were recognized by the program by passing various characteristics for a given symbol through a decision tree. For a delete or substitute operation, for instance, the user drew a line through the text to be deleted. The system deleted this line, blinked the indicated text for verification, separated the text by opening a blank line, and inserted a cursor, enabling new text to be typed in from the keyboard. For a transpose operation, the user simply used the familiar transposition mark. With the existence of data tablets with built-in microprocessors dedicated to this recognition task [IMAG81], we may expect to see this technique used in commercial editing products in the near future.

The major innovations of the 1970s in text handling and the user interface, specifically the Bravo editor [LAMP78] and the Smalltalk environment, took place at Xerox's Palo Alto Research Center (Xerox PARC). These systems demonstrated the expressive power of blending text and graphics on a high-resolution, bit-mapped, raster graphics screen, using a dynamic graphical interface provided by a dedicated personal computer. Editing was done by selecting items on the screen, using the mouse as a pointing device. These systems were also the first *interactive editor/formatters*, in which the user's text was displayed on the bit-mapped screen in a facsimile of the typography and layout of the final document, as the document was being input or modified. For the first time, the user was given a notion not only of the up-

to-date content, but also of the up-to-date form of the document.

Current research in the field of editing is focused upon several overlapping areas. One is that of providing a consistent, editor-based interface throughout a computer system [GOLA79, LANT80, FRAS80, APOL81, STAL81, TESL81, GOLA82]. This allows many common functions, such as renaming files, searching through directories, and debugging programs, to be performed as editing operations. For instance, to rename a file, one would type over the old file name in a listing of available files that would appear on the screen; in debugging a program, one would be able to edit the values of displayed variables. Other research topics include generalized structure editors, powerful syntax-directed editors with program-tracing capability, and interactive editor/formatters. Research in these and other areas is discussed more fully in Part II, Section 1.

3. FUNCTIONAL CAPABILITIES

In this section, we take a more detailed look at interactive editors, examining their functional capabilities from a user perspective.

3.1 Traveling

User-specified traveling commands cause the *current editing* and *viewing pointers* of the editing and viewing components to be moved, resetting the filters that extract the editing and viewing buffers.[8] Traveling ranges from simple motion such as moving to a subsequent screenful of data, to more complicated movements such as pattern search and tree or directed-graph traversal.

Early systems, and even many of today's word processing systems, are oriented toward transcription of editing changes from hard copy. They therefore provide relatively unsophisticated traveling mechanisms. For on-line authoring, however, a system should be oriented toward browsing, studying, and organizing as well as toward composition and revision. In such a system, traveling flexibility and power are mandatory, as the NLS experience has shown [ENGE68].

3.1.1 Simple Movement

Editors commonly support two types of *intrafile motion*: *absolute* and *relative*. An absolute specification indicates a destination independent of the current position, while a relative specification indicates the destination relative to the current position. Generally, the current pointer is maintained internally to point to the corresponding current position in the data structure; in editors with pointing devices, the cursor usually points at this position in the visible text with the current pointer in lockstep. The user can move the editing buffer and viewing buffer from this position by issuing simple commands. In IBM's line-oriented CMS editor, moving the editing buffer and the viewing buffer five lines ahead in the document would be accomplished by typing

```
next 5
```

In line-oriented editors, line numbers can be *fixed* or *varying*. In editors with fixed line numbers and numeric labels provided by the user or the system, the user can easily specify an absolute goto to that number. As an example, the editor portions of BASIC interpreters use fixed line numbers specified with even intervals, such as

```
00010 FOR I = 1 TO 10
00020   J = I * I
00030 NEXT I
```

To add a print statement after the increment statement, one could simply type

```
00025   PRINT I, J
```

and the editor would put the new line between lines 20 and 30, as indicated by its numeric label.

In contrast, editors with varying line numbers keep track of a line's position internally as an offset from the top of the file. When a new line is added, the internal line numbers of all lines beneath the new line are incremented by one. Since the line numbers change dynamically, the deletion (insertion) of a line near the top of the file causes all the line numbers below it to be decremented (incremented) by one. Because of this, editing by specification of line

[8] To simplify the remainder of our discussions, we group both pointers together under the term "current pointer."

number is best done from the bottom of the file upward, since insertions or deletions will typically only affect the lines at and below the current pointer. This is a restriction palatable (if at all) only for transcription from hard copy, not for on-line revisions.

As we have seen, the user of a display editor travels by pushing function keys such as

+LINE	or	−LINE
+WORD	or	−WORD
+PAGE	or	−PAGE
+1/2 PAGE	or	−1/2 PAGE

that effectively change the contents of the editing buffer and the viewing buffer. Additionally, the editing and viewing buffers can be moved left or right by a specified number of character positions (columns) to allow editing of wide lines, as in the following command:

```
50 LEFT
```

Top and bottom (of file) are two very common, highly functional traveling commands.

Many editors have *mark* or *saved-position stacks* or *rings* that record locations in the file. Some maintain an implicit mark ring, automatically storing the location each time that the user makes a selection for a command, and allowing the user to retrace his or her steps. Others have explicit mark buffers; the user uses the save position command to store the current location in any number of buffers and the goto position to return to a saved position. Additions or deletions in the file will throw off the exact saved location if the editor only saves a character-pointer or line-pointer or other varying index, but will typically leave the user in a location close to the original marked location.

Systems like NLS and FRESS allow character string *labels* to be inserted in the text; these move as the adjacent text is moved. The user need not know the location of a piece of text, whether relative or absolute, in this system, but, for example, can travel instantly to the text labeled "casestudy" with the command goto-label casestudy. In the case of program editors, identifiers in procedure declaration statements can be used as implicit labels, as in the EMACS *tags* facility explained in Part II, Section 1.

Of course, many target-dependent traveling operations exist. In document editors, for example, simple movements are provided to bring a user to the beginning of a section or chapter. In a program editor, a traveling command might take the user to the next syntactic element. More specific examples are provided in Part II, Section 1.

3.1.2 Pattern Searching

The above methods of traveling are **position** dependent: goto line 7, go-back 15 lines, or goto-label sec5. Almost all modern text editing systems additionally allow a **content**-dependent specification of location: *pattern searching*. A pattern searching command (usually called search, find, or locate) generally changes the editing and viewing buffers so that they contain the pattern that is being sought.

Since typing an entire pattern is a tiresome operation, specification aids have been developed. The familiar " . . . " (*ellipsis*) construct can be specified to abbreviate a long pattern by indicating simply a few characters of context at the beginning and end of the pattern. Thus, one can locate the text string:

```
Now is the date for some good men
```

with the command:

```
locate /Now . . . men
```

Regular expression context patterns extend the ellipsis concept to more powerful pattern-matching capabilities, for instance, matching a pattern that does or does not contain a particular set of characters, matching a pattern only if it occurs at the beginning of the line, or matching a pattern regardless of the case of the characters (known as *case-insensitive pattern matching*).

3.1.3 Interfile Motion

The ability to travel between two or more files is extremely useful in on-line authoring. At the least, the user must be provided with commands to switch between two files. (This is much more useful, of course, if the

two files can be displayed in two windows on the screen.) More advanced editors maintain a circular list of the files visited; the user can travel from one to the next or the previous in an ordered fashion, or in some cases choose any of the files from a display of the circular list.

Interfile travel generates several new problems. An editor may need a more intricate internal data structure and associated editing and viewing buffers for each open file to allow multiple files to be active at once. If the editor has the ability to map file viewing buffers to multiple windows, but can only keep track of one file at a time, a file must be saved each time the user switches windows; this discourages on-line browsing. As well, attributes can be kept for each file so that when a user is editing a LISP program, the editor matches parentheses and automatically indents lines. When the user returns to a textual document, the editor should no longer do so.

3.1.4 Hypertext and Trails

The above types of travel are usually employed by the user **during** the editing of a document, typically to get to the next place in the file where an edit is to be performed; the paths of travel are not stored from one editing session to the next. Occasions do arise when the user wants to set up (semi-) permanent paths or links within a document or between documents. The motivation for such text links is well stated in a seminal article by Vannevar Bush, who envisioned the "memex" device for authors and readers in 1945:

> The human mind . . . operates by association. With one item in its grasp, it snaps instantly to the next that is suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain. It has other characteristics, of course; trails that are not frequently followed are prone to fade, items are not fully permanent, memory is transitory. Yet the speed of action, the intricacy of trails, the detail of mental pictures, is awe-inspiring beyond all else in nature . . . Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and, to coin one at random, "memex" will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility . . . when numerous items have been thus joined together to form a trail, they can be reviewed in turn, rapidly or slowly, by deflecting a lever like that used for turning the pages of a book . . . It is exactly as though the physical items had been gathered together from widely separated sources and bound together to form a new book. It is more than this, for any item can be joined into numerous trails. [BUSH45, pp. 106–107]

Hypertext editing systems are the modern-day incarnation of Bush's precomputer memex. Hypertext, a term coined by Ted Nelson is "the combination of natural language text with the computer's capacities for interactive branching, or dynamic display . . . a nonlinear text . . . which cannot be printed conveniently . . . on a conventional page" [NELS67, p. 195]. Simply, hypertext is defined as nonsequential writing. Hypertext systems allow the user to construct arbitrary links from any chosen point in a document to any other point in that document or in any other document in the user's domain. Menus of such links can be set up to provide a branching text that constantly evolves to provide a "dynamic and plastic structure" [ENGE68]. A batch formatter can follow keyword-specified links to compile the text needed to produce a document. An interactive user can select links by keyword specification or by menu selection to lead to desired text, typically displayed in a separate window. In NLS, links can also be accessed via complex specifications of structure and/or content. One can specify, for example, "follow the first link containing the string 'edit' in any third-level statement in the hierarchy, starting the search at the current first-level statement in view."

In the simplest sense, a hypertext document can consist of a "table of contents" with on-line links to files containing each chapter, each of which can link to sections. More advanced forms of hypertext include the use of links, rather than the on-line equivalent of standard footnotes, to point to the actual material referenced, allowing instantaneous access to cited material. Thus a document can be browsed on-line, and predesigned-but-optional diversions can be set up along trails that readers may find interesting. Of course, authors and

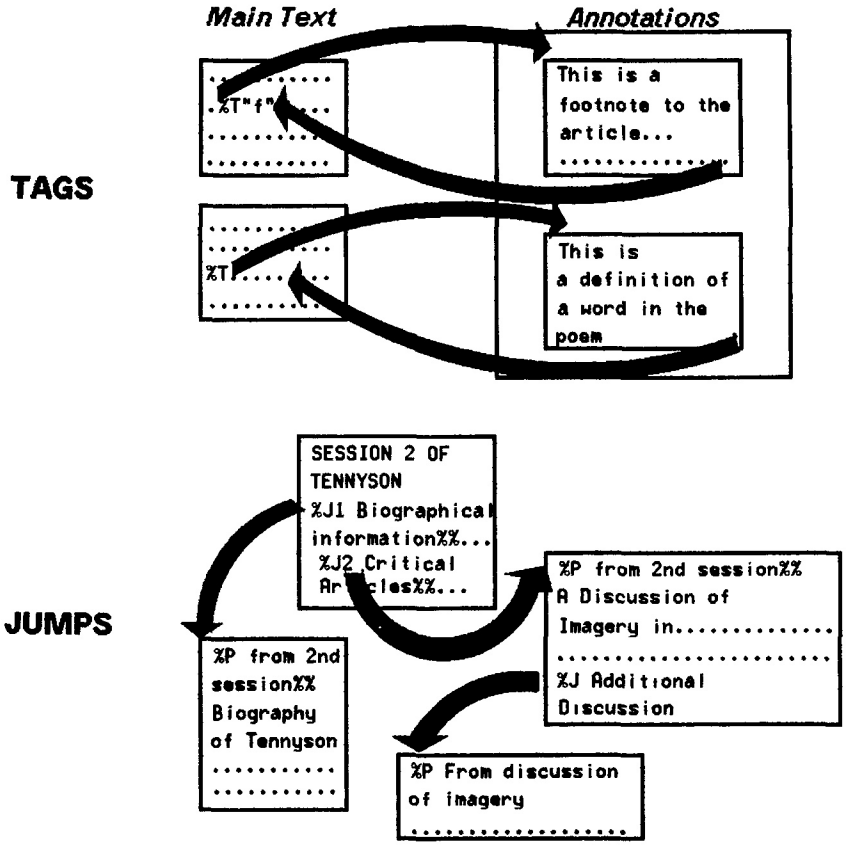
Main Text
Annotations
TAGS
This is a footnote to the article...
This is a definition of a word in the poem
JUMPS
SESSION 2 OF TENNYSON
%J1 Biographical information%%...
%J2 Critical
%P from 2nd session%% A Discussion of Imagery in.....
%J Additional Discussion
%P from 2nd session%% Biography of Tennyson
%P From discussion of imagery

Figure 4. An example of hypertext in FRESS. *Tags* and *jumps* are two of FRESS's hypertext elements. A tag T points to a single element (such as a footnote) in an "annotation space"; the user views the annotation but remains in the main text. A jump J indicates a path to another part of the current document or to another document. By taking a jump, one travels, changing the editing and viewing buffers of the document. In the new context, one can edit, take another jump, or take the reverse of a jump (labeled P for *pmuj*, jump spelled backward), which returns to the unique source of the J that brought the user to the P in the first place. Multiple jumps to the same text would have multiple P labels so that the user could distinguish the sources. Other system commands allow the user to retrace steps on a session basis, that is, to return from all previous jumps with explicit return commands.

readers are free to create links to new files or to *annotation spaces* in which they may comment on what they are reading, thereby enriching the trails for others. A hypertext system may allow the user not only to see links to other places **from** the current position, but also to see that there are links from other places **to** the current position [VAND71a] (see Figure 4). The browser is then free to jump backward to see from what document this link was issued. A hypertext document is a bidirectional network of associative trails, a constantly evolving directed graph structure of text nodes that readers and authors can traverse. Catano [CATA79], Robertson et al. [ROBG79], and Herot et al. [HERO80] provide examples of the use of this arbitrary directed-graph structure for documents. Feiner et al. [FEIN82] use the hypertext concept in the context of graphics-oriented "electronic books" (see Figure 5).

3.2 Viewing

To provide a basis for editing and browsing, and to provide feedback after an editing command is executed, the viewing compo-

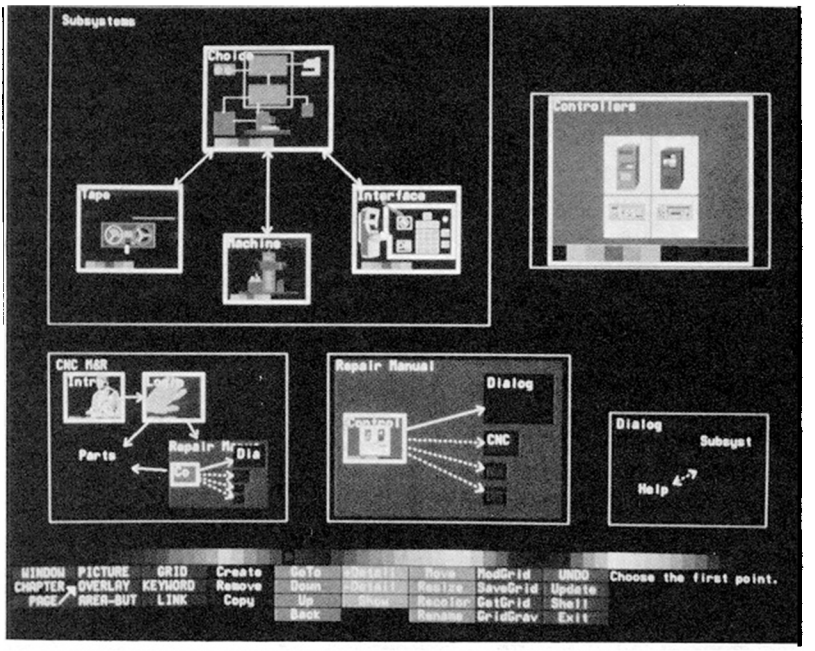

Figure 5. Electronic book layout. A multiple-window display in the document layout system designed for authors and readers. The author may traverse the document's structure independently in each window and make links between chapters and pages in different windows. The author may also create, resize, and remove windows as needed, and revise the level of detail of the pages and chapters shown to control the amount of information presented.

nent creates an up-to-date view on the display. The viewing operations are of three sorts: (1) filtering operations in which appropriate portions of the raw data structure of the file are selected for the viewing buffer, (2) operations that format these filtered data to produce an ideal view, and (3) operations that map this ideal view to a window or page on a physical output device.

3.2.1 Filtering

Editors that are especially oriented toward browsing (e.g., NLS and FRESS) usually provide facilities to alter the *viewing specifications* (*viewspecs*), allowing the user to control the filtering operations. Viewspecs can be used to control any number of selection and viewing parameters (parameters controlling what is to be displayed and how, respectively) to effect *information hiding* and on-line formatting. For instance, NLS allows the user to indicate what levels of detail in a hierarchy are to be displayed, as well as how many characters per statement and even which statements, on the basis of content specification (e.g., "fill the window with the first 50 characters of as many of the statements that lie between Sections 1.3 and 1.9 as possible, up to and including the third decimal level"). Additionally, the viewspecs control whether the selected text is to be shown normally or in indented outline form, with or without section numbers, with or without format codes, or in ragged-right or right-justified format. *Display keyword* or *password viewspecs* allow the user to see pieces of text only upon the presentation of the proper key; this facility can be used to protect sensitive materials and to reduce the clutter of on-line presentation by presenting only (potentially) relevant information. Similarly, a syntax-directed editor can allow the user to turn off the display of declarations or comments in a program.

Since the view is a mapping of the internal data structure onto the user's display, external factors such as noise on the communications line or a received system mes-

sage can corrupt the displayed view, while the internal data structure remains intact. A system often supplies a refresh display function to restore the appropriate view from the internal data structure.

3.2.2 Formatting[9]

The simplest method of formatting the filtered viewing buffer contents is the "null" formatting: the text is shown exactly as it is stored in the internal data structure. Typically this is not sufficient for the user. In simple editors, text is stored essentially as consecutive characters on disk, with special characters indicating tabs and physical ends of lines. A simple formatting routine is needed to map stored lines to a screen image, inserting logical carriage returns and/or line feeds where appropriate. In many editing systems, especially stream editors, the formatting routines must make on-the-fly decisions about where to break lines that do not fit on one screen line [KNUT79, ACHU81]. For program editors, an intelligent formatting routine might include automatic indentation of program constructs.

In early text processors, *formatting codes* (such as .pp for paragraph, .in 5 for indent five spaces, .ce for center) were typed in as literal text and subsequently compiled in a batch-formatting pass to produce formatted pages; no on-line feedback was available. Later *soft-copy* or *proof-copy* facilities were made available to display (but not to allow editing of) monospace (line-printer quality) output on the alphanumeric terminal. With high-resolution point-plotting and raster displays, even proportionally spaced typeset text could be previewed outside the editor as the result of a separate batch formatting pass. The next major step in the formatting field was the creation of the interactive editor/formatter. Today most editors, especially commercial word processors, instantaneously display the results of commands with local effect (such as indent, tab, embolden, and center) as they are entered or changed. In the newest generations of word processing systems the idea of on-the-fly formatting has been taken to the next logical step: they allow essentially all formatting, including hyphenation and pagination, to be done on the fly. Interactive editor/formatters make possible an especially useful view in which all operations on the document take place immediately on a displayed facsimile of the printed page, thus giving instant user feedback. Other optional views may include the facsimile document along with ancillary windows in which the typesetting codes controlling the formatting are displayed (see the description of the Xerox Star in Part II, Section 1), or with in-line "tags" describing the various document elements that appear on the screen (see the description of ETUDE in Part II, Section 1).

With hardware such as the personal work station becoming prevalent, users will be less patient with traditional, tiresome, monospace characters on 80 × 24 CRTs, and will demand that display output make use of the many hard-copy information-coding techniques that improve reading efficiency. These include proportionally spaced text, font changes, highlighting, complex page layout, and even embedded equations and graphics. Thus on-line formatting for display (soft copy) and off-line formatting for paper (hard copy) will become more similar, especially when bit-map raster displays approaching the 200-points-per-inch resolution of inexpensive (less than $10,000) electrostatic or laser printers become commonplace. (Even before this becomes a reality, we see that computerized typesetting, once reserved for special documents or for final copies of a document whose drafts were produced on line printers or typewriter terminals, is now within the reach of many commercial and academic installations for routine work.) Editors must therefore make it possible for the user to specify sophisticated formatting effects shown on-line during the editing phase. Good examples of systems providing interactive typesetting can be found in several works in the literature [LAMP78, HAMM81, SEYJ81, SMIT82, XERO82].

3.2.2.1 Formatting Commands. Approaches to the specification of formatting are numerous. In some systems, primitives

[9] We do not attempt to cover the entire field of formatting here, but simply point out salient features of formatting directly tied to the editing task. We refer readers to FURU82 for a thorough survey.

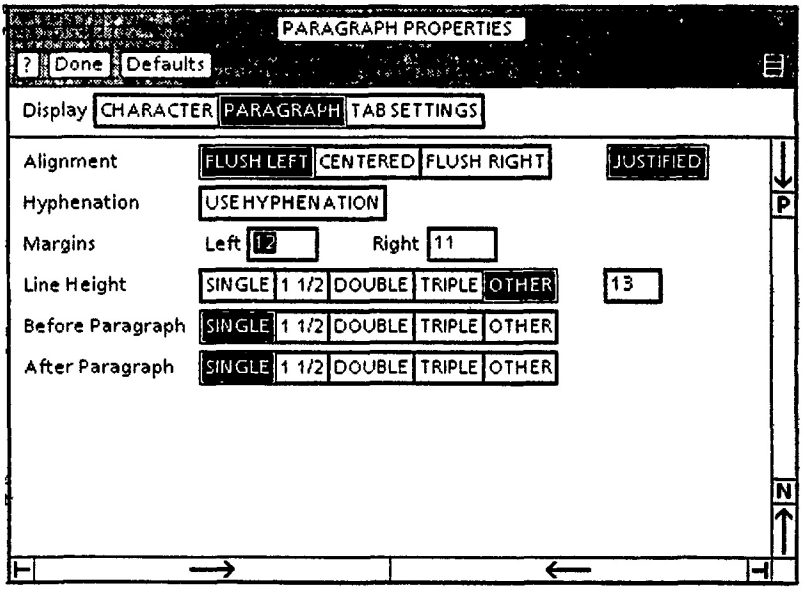

Figure 6. Xerox Star property sheet. The Xerox Star property sheet allows the user to edit attributes of various elements. The paragraph property sheet lets the user change alignment, spacing, and hyphenation by pointing at graphical buttons on the screen. Selected attributes are shown in reverse video. (From XERO82, courtesy Xerox Corporation.)

provided as part of the interaction language allow the user to specify formatting operations on designated elements. For instance, the use of the center command in the editor might actually insert the appropriate spacing for centering and alter the internal representation of the document. Alternatively, this type of specification may be used to generate a low-level representation such as a formatting (typesetting) code that is stored in the file. In some systems these are invisible to the user and a mechanism must be provided to change these low-level representations using high-level editing commands. For example, many word processors provide a *tab rack*, an on-screen simulation of a typewritter's margin and tab controls, that contains the current margin settings, font styles, tab stops, and so on. The user can change these settings by simply moving the cursor into the tab rack and editing the attributes. In Xerox's Star, alternatively, every element (from a single character up to the entire document as a whole) has associated with it an optionally displayable *property sheet*, which simulates a preprinted form or checklist. The user can examine the property sheet at any time and fill in or change the stored formatting options for any element (see Figure 6).

In other editors, the formatting specification is entered as a formatting code in the same manner as "normal" (literal) text. In some systems, these codes must be entered on separate lines to distinguish them from literal text; in other systems a special character is used as a delimiter so that the codes can be imbedded in the normal text stream; two delimiters in a row then designate a literal delimiter character.

Regardless of how they are indicated, the specifications are one of two types: *procedural* or *declarative* (*markup*). In a procedural specification, the author indicates the exact operations to be done to effect the formatting choices (e.g., skip two lines and indent three spaces). Conversely, in a declarative system, *tags* are used to identify elements of the document, such as items in a numbered list, paragraphs, chapter headings, and running heads. A separate facility stores the actual formatting attributes for the particular elements, and the formatter recognizes the tags and uses the tag facility

(usually a set of formatting macros or a database of formatting information) to determine **how** to format any part of the document. These tags allow the same document to be formatted in different styles or for different output devices by simple parameterization: for instance, the user could format a *Computing Surveys* document by specifying the tag ⟨Style = Surveys⟩ at the beginning of the document and for *Communications of the ACM* by specifying the tag ⟨Style = CACM⟩. Similarly, the user could format a document for the line printer by specifying the device tag ⟨Device = lpr⟩ and for a high-quality typesetter by specifying the tag ⟨Device = typesetter⟩. *Style sheets* containing a skeleton of tags for a particular document, for example, an on-line corporate memo that can be filled in by users, allow standardization of documents for a community, much like preprinted stationery or pads. Declarative languages provide less complicated user specifications, and often less complicated editor interfaces for on-line formatting; the user need not supply detailed formatting commands, and need use only high-level structural indicators as tags. Reid's Scribe [REID80a, REID80b, REID80c] is a popular declarative language and document compiler. IBM's GML [GOLC81] is a formatter-independent declarative language. Hammer et al. [HAMM81], Walker [WALK81b], and Chamberlin et al. [CHAM81] discuss editors that make use of the declarative technique (see Part II, Section 1).

3.2.3 *Window Manipulation*

In editors with multiple windows, the user must be able to specify and alter the placement of the windows on the display. Though the system can allocate windows—a single window could occupy the entire screen, two windows, half the screen each, and so on—the system can allow the user to specify the windows, either by typing coordinates or, more likely, by defining the bounds with a pointing device.

3.3 Editing

Our discussion of the editing component focuses on two major concerns: specifying the operands of an editing command and the operations themselves.

3.3.1 *Specification of Scope*

The range of a document used as an operand for an editing operation is called its *scope*. The user typically thinks of the scope not in terms of "raw" text, but rather in terms of logical elements that can be both edited and traversed. These elements are also known as units, nouns, objects, and structures, and are often a function of the internal representation of the target data. In line editors, for example, manuscript text may be stored as sequential lines and edited and traversed on a line basis. In stream editors, the text may be stored as a one-dimensional, indefinitely long stream of characters, broken into discrete lines by formatting routines for viewing purposes. Programs may be stored in textual form or in some intermediate form such as a parse tree or abstract syntax tree.

In addition to logical elements that match the internal representation, editors provide a variety of elements corresponding to user-level abstractions, often grouped in order of increasing scope: characters, words, lines, sentences, paragraphs, and sections are typical for text. In addition to these standard system-provided elements, users can also define arbitrary *regions* or *blocks* that exist only for the duration of a given operation. A third class of elements may be a (partially overlapping) set of abstract document components such as sections, headings, titles, running headers, footers, and numbered lists. These are typically defined in terms of formatting conventions and are available for explicit manipulation only in the most recent editors.

The goal of scope specification for all these elements is to approximate within the limitations of the computer interface the ways users would manually gesture ("delete **this**, move **that** over **there**"). As we have seen, in line editors the user establishes the current line (by specifying its line number or by traveling to it via scrolling or pattern searching) and then specifies the scope (by typing either a context pattern within that line or an integer to indicate a group of one

or more discrete lines whose first line is the current one). In the UNIX *ed* editor the command

```
s/Julius/Brute/
```

substitutes the word "Brute" for the scope "Julius" when the user is at the line

```
Et tu Julius!
```

In stream editors, of course, the pattern matching is not limited to a current line but has potentially the entire file as its domain. In display editors, the user specifies the scope by driving a cursor with a locator device to define the appropriate element(s), in a manner more directly analogous to the manual technique of pointing.

There are several techniques for specifying a scope, some applicable to a typing-oriented interface, some applicable to a pointing-oriented interface, and some applicable to both.

In the first technique, the user simply selects an element—a character, word, line, or paragraph, for example. The user may *extend* this selection by selecting another element of this same type; all elements between the starting point and the other selected point (inclusive) would be selected. This is typically done using the ellipsis or regular expression facilities in a textual interface, and direct cursor positioning in a pointing interface.

In the second technique, the user adds modifiers to the commands. For example, having established a location in the text with context or cursor specification, the user can complete or modify the scope of a delete operation with word to form delete word or with a numeric parameter to form delete 3 words.

In the third technique, the user again establishes a starting selection at the smallest element (e.g., a character) and then *adjusts* the selection. For instance, by using the mouse in Xerox's Star, a character can be selected as an initial scope and this scope adjusted to include successively the word, the line, the sentence and the paragraph containing that word (see Figure 7).

In a hierarchical structure editor, selection and adjustment can easily be used to traverse the hierarchy, as in select node and extend to left child.

3.3.2 Editing Operations

3.3.2.1 Creating. Text is inserted into a computer-based document with a text input device. To enrich this hardware, special software is frequently supplied. Editors often provide automatic *wordwrap*, which eliminates the need for typing a carriage return at the end of each line. When the editor senses that the word currently being typed has exceeded the right margin, it breaks the line at the first blank space before the overflowing word and automatically pushes it, followed by the cursor, to the next display line.

Display editors generally offer one of two kinds of input styles: *typeover mode* or *insert mode.* In typeover mode, each typed character replaces the character at which the cursor is pointing. For adding characters, such editors must supply insert character or insert line functions that open a blank character or a blank line at the cursor position. In insert mode, each time a user types a character, the character at the cursor position and all those to its right are shifted right one character and the typed character is inserted at the cursor position. Thus insertion can always be done without any commands, while replacement requires a command such as delete character or delete line before or after the insertion. Many editors allow the user to toggle between typeover mode and insert mode.

Other creation techniques include the capability of *embedding* files or parts of files into the document being edited, thus making it possible to recycle previously created material. The user can create *boilerplate* documents, as is often done with proposals, contracts, and specifications, by using bits and pieces from the entire domain of user files on the computer. Form letters can be created by embedding consecutive addresses from an address file at the top of a generic letter file.

3.3.2.2 Deleting. The delete command requires the user only to select the scope of the operation. Since deletes are obviously dangerous commands, some systems re-

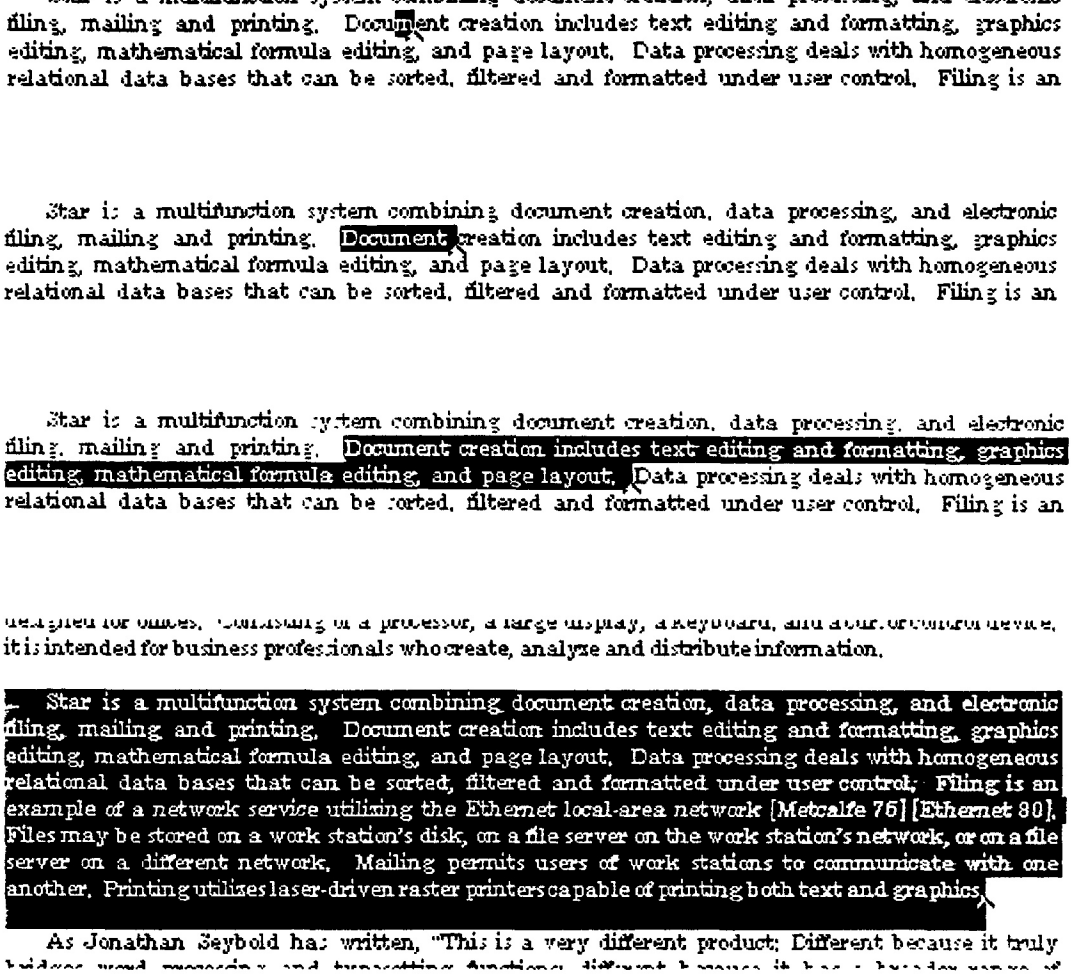

Figure 7. *Selection in Star. One click of a mouse button selects a character, the second click adjusts the* selection to be the word containing that character, and the third click adjusts to the sentence containing that word.

quire confirmation before actually completing the operation. Other systems allow the user to undo commands, making the deletion operation reversible. For the delete command, as well as the copy and move commands described below, many systems provide *delete buffers*. This allows the deleted elements to be placed in "limbo" so that they can be used later as the objects of put (also called insert) operations, which put the elements from the delete buffer back into the text. To move a paragraph, for instance, the user first selects it and specifies the delete operation; the editor then deletes the paragraph and places it in the delete buffer. To put the paragraph back in somewhere else, the user indicates the desired destination and specifies the put operation. Often, systems provide multiple, named delete buffers for more complex manipulations. Berkeley's *vi* editor [JOY80b], for instance, has a *buffer stack*, which keeps track of the last nine pieces of text that have been deleted. EMACS [STAL80, STAL81] has a *kill ring*, a circular list containing the last eight blocks of text that were deleted. The user can "walk around" the previous kills in this ring until the desired text is found, with the walk always leading back to the starting point. Usually

delete buffers and kill rings are saved while performing interfile editing, so that text deleted from one file can be inserted into another easily.

3.3.2.3 Changing. Many of the changes made to a document are corrections of typographical and other minor errors. The simplest change is the replacement of one letter with another. In typeover mode of a display editor, the correction is made by simply typing the new character over the erroneous one. In insert mode of a display editor, the correction is made by typing the new character and then using a delete-char command to delete the old character, which has been pushed to the right by the insertion of the new character. Similarly, changing a word in typeover mode simply involves typing over the erroneous word. However, since the replacement word may be shorter (longer) than the original word, a delete (insert) character function may also be needed. In insert mode, changing a word would be done entirely with the implicit insertion combined with delete-word functions.

In editors without cursor keys, the user needs a method of specifying what character(s) to replace. These editors have a change or substitute command, as shown previously, that takes as arguments both the scope of the change and the replacement string.

Because of its speed, the computer can offer the facility of global operations—operations that take place uniformly throughout a document. The global change command allows the user to specify a pattern to be found throughout a document and a replacement string to replace that pattern wherever it appears. In some systems, this command may do its work but not indicate what changes have been made. In others, a count of the number of changes is given. In others the pattern string is highlighted each time it is found, and the user is prompted to indicate whether or not the change is desired. Column-dependent changes are useful for editing programs or tables.

The transpose command is a special-purpose change command. In EMACS for example, the CTRL-T command will exchange the character at which the cursor points with the one directly to its left, and similarly, the META-T command will do so for words.

3.3.2.4 Moving. The ease with which blocks of text can be rearranged is one of the great advantages of interactive editors. To move a block of text the user specifies a source (the scope of text to be copied) and destination (the place where the text is to be copied). In an editor in which the scope is specified by pointing, the user defines the source by selecting the beginning and end of the text to be moved, and then defines the destination by pointing to the location at which it should be placed.

In line editors and context editors, where such a physical sequence of steps is impossible, the user must resort to textual specification. For example, the misordered poem by Frost

```
Two roads diverged in a yellow wood,
And looked down one as far as I could
To where it bent in the undergrowth;
And sorry I could not travel both
And be one traveler, long I stood
```

could be ordered properly with a typical line editor by traveling to the line "And looked down one as far as I could" and specifying

```
move 2 down 2
```

This would temporarily delete the following two lines, including the one currently being pointed to, thereby moving the current pointer to the line following the last line deleted, then move it down two more lines, and insert the temporarily deleted lines where the current pointer now points.

Note that with most line editors, the user is severely limited to moving integral numbers of lines, rather than arbitrary regions. In a context-driven stream editor this specification is a bit easier:

```
move/And looked . . . undergrowth;/stood/
```

Rather than specify relative line numbers, the user now specifies context patterns.

Note that systems with delete buffer facilities may have no need for a special-pur-

pose move command, since, as discussed in the previous section, one can delete the desired text and reinsert it in the appropriate place with a put command.

3.3.2.5 Copying. A copy command is simply a move command in which the source text is not removed after the destination text is in place.

An alternate strategy for the copy command is to have a pick command that stores a selection of text in a *pick buffer*. This is analogous to using the delete buffer described above, with the important exception that the picked text is not deleted. Now a copy of the picked text is in limbo in the pick buffer and can be reinserted in the text with the put command. If the put command does not erase the contents of the pick buffer, this facility can be used to make multiple copies of selected text quickly.

3.4 Miscellaneous Capabilities

Commands grouped under this heading are not directly involved with manipulating text, but rather with assuring the integrity of a user's work and with making the user interface more powerful and helpful.

3.4.1 Reliability

3.4.1.1 Backup Capability. To minimize the possibility of the accidental erasure or destruction of a document, editors often have *backup* capabilities. One strategy is to give the user *work files*, copies of the actual files, to work with, saving them as the actual files only when the user exits or when an autosave feature copies the work files as the actual files after a (user-specifiable) number of keystrokes or command executions. A similar strategy is to make automatically a backup copy of the files as they exist when the editor is invoked; the user is given the actual files to edit, but can always make the backup copy become the actual file in the case of a system crash or editing mishap while editing the actual file. An abort command allows the user to scrap an editing session and return to the backup copy that contains the file as it was when the editing session began.

3.4.1.2 Undo Facility. The undo facility is a critically important, time-saving, and unfortunately not yet universal feature. The most basic version allows the user to undo the last command entered. More useful systems have an n-level undo stack that allows the user to undo commands n levels back (sometimes to the beginning of the session or even back to previous sessions). The undo feature frees the user from the burden of making sure that each command does exactly what is wanted by guaranteeing that any result can be undone. The general undo facilitates risk-free experimentation, which is especially important for on-line composition and organization. Some systems [ARCH81] provide an undo–redo facility, which allows the user to undo operations and then redo them at the push of a button. Another way of presenting undo, rather than as a complete backtracking of a session, is simply as a command that performs the inverse of the last specified command. For instance, after one performs a delete, the undo command would essentially perform a put. Issuing undo again would then perform a delete. Smalltalk-80's cancel/accept pair allows the user to accept the editing state at a point in time and experiment freely, knowing that a single cancel command will return the document to the accepted state.

3.4.1.3 Cancel Facilities. The cancel command allows the user simply to cancel the command that is currently in progress. This is of great importance if time-consuming operations such as pattern searching have been specified erroneously. When commands are not queued and type-ahead is not in effect, specifying a new command while a previous command is still executing often implicitly cancels the executing command. Alternatively, especially when an operation is simply a traveling command rather than a command that makes extensive changes to the internal data structure, specifying a command while one is executing could cause feedback from the first command to be canceled. For example, if the user presses the + PAGE key in a screen editor, and, while the new page is being displayed, the user presses + PAGE again, an intelligent system would not first complete the display of the first page and then replace that completely with the second page, but would cancel the display of the

```
savetemp=1
bufinfile=.bbbuf
bufoutfile=.bbbuf
filename=macrofig.me
curcrypt=0
curstream=1
curtabin=0
curtabout=0
curindent=1
curline=19
curchar=43
curscreen=0
curmargin=0
curwindow=0
curreadonly=0
curlanguage=roff
fileid=10
Pfilename=dopstitle.tex
Pcrypt=0
Pstream=1
Ptabin=0
Ptabout=0
Pindent=1
Pline=17
Pchar=6
Pscreen=7
Pmargin=0
Pwindow=0
Preadonly=0
Planguage=
Pfileid=10
Qfilename=/usr/info/b.help
Qcrypt=0
Qstream=0
Qtabin=0
Qtabout=1
Qindent=1
Qline=0
Qchar=0
Qscreen=0
Qmargin=0
Qwindow=0
Qreadonly=0
Qlanguage=
Qfileid=2
Asearch=shar
Arsearch=^[WM]
Afsearch=
Aexit=
Abfsave=
```

Figure 8. A *bb* control file. The control file saves information from one editing session to another. Here, for example, the bufinfile and bufoutfile variables indicate where the pick and delete buffers are to be saved from session to session. Curline and curchar indicate where in the document the cursor was pointing when the session was ended. Curscreen indicates which line is at the top of the screen. Filename indicates what file was being edited at the time the session was ended. The variables beginning with P and Q keep track of the same information for multiple files.

first page and proceed to display the second page.

3.4.1.4 Keystroke History. The *keystroke history* is a powerful reliability mechanism that keeps a copy of every keystroke (both command and text) that the user has specified since the current editing session started. If the system crashes, the user is provided with mechanisms to run the keystroke history file against the old copy of the edit files as if the commands were being typed in. This history file can also provide the basis for the undo command.

3.4.1.5 Context Saving. A useful feature provided by some systems is the ability to save editor attributes and parameters from session to session. In *bb*, for example, a control file (see Figure 8) saves data about the initial cursor position when the editor is invoked, the file to use, and enough other information that the user can simply invoke the editor, without any parameters, and continue a session exactly where it ended five minutes or five days ago. In more advanced systems, a *checkpoint/restart*, *snapshot*, or *workspace* facility allows the recording and subsequent restarting of the entire editing environment, including complete internal data structures. This is common in programming languages like LISP, APL, and Smalltalk-80.

3.4.2 Ergonomics

3.4.2.1 Repetition. A repeat command, which reexecutes the command last executed, is a simple and convenient facility. Another alternative is to have the user apply a new selection and then reissue the command. Another is to "bring back" the last command and allow the user to edit it before reexecuting it.

3.4.2.2 On-Line Documentation/Help Facility. An on-line *help facility* is extremely important to new and occasional users, as well as dedicated users who do not use all parts of the system regularly. The

help facility can provide an expanded explanation of an error message, a short summary of a command syntax, or perhaps complete access to an on-line version of the manual. Some systems create a separate help window, allowing the user to have access simultaneously to both the help information and to the document being edited, rather than forcing the user to leave the document and lose the information on which the user sought help in the first place.

3.4.2.3 User Feedback. Feedback is a vital part of the editing process; it is necessary for specifying operations, for specifying scopes, and for showing the results of an operation in the updated view. The last kind of feedback is provided out of necessity, but the first two categories are not always given due consideration by editor designers.

In editors with typing-oriented interfaces, echoing the typed command provides immediate feedback on both operation and scope. In function-key interfaces, a button push provides no inherent feedback. If no supplementary feedback were supplied, the user would have to rely on examining the results of the operation to see if the specification was correct; by this time, it would often be too late to reverse the results. Thus feedback techniques, such as highlighting a selection in progress (with such hardware-supported techniques as brightening, underlining, or reverse video) in display editors, or highlighting the menu items as they are browsed through in menu-oriented interfaces, are vitally important. In non-display-oriented systems, a summary of the command that is to be executed or has been executed is useful. This feedback might consist of displaying a condensed English-language message such as "move 'red fox . . . dog' after 'The quick'." If an editor has undo facilities, this type of condensed feedback of results is useful but not necessary, since the user can afford to make mistakes or experiment.

Other audio and visual cues aid the user at a little cost in efficiency or in implementation. Beeping to signal errors usually ensures that a user does not miss the occurrence of an error. Programmable cursors allow the cursor to take on different symbolic forms depending upon what the user is doing. Time-consuming operations require intermittent feedback so that the user can be satisfied that the system is still working. Newswhole [TILB76] uses a Buddha icon to remind the user to be patient, while the Xerox Star uses an hourglass icon to indicate that it is busy. A status line at the top or at the bottom of the screen, indicating the current position in the document, the name of the file, any modes that might be set, and other such information, is an easily implemented method providing positive feedback.

3.4.3 Customization

3.4.3.1 Profiling. A *profiling* facility allows the user to "tune" the editor environment at invocation time. This allows important or preferred environment settings to be handled automatically and removes the need for all users to accept a common default. Some editors allow the setting of simple state variables such as number of backup versions and number of modifications before saving the file on disk. Others allow each user to reconfigure major parts of the interface, such as redefining function key bindings.

3.4.3.2 User-Defined Commands (Macros). Editing systems often allow the user to define *macros* or *editing scripts* (super-instructions) based upon the system operation repertoire using an editor *macro language.* The user can thus package under one name sequences of commands that are often executed as groups (see Figure 9). In some systems, these commands prompt for or simply accept parameters (operands) and even provide conditional execution, for maximum power and flexibility. Function-key editors often have *keystroke macros*; the system "captures" a set of keystrokes typed in by the user and can then repeatedly execute those keystrokes as if they were one command. Some keystroke macro systems even allow the user a form of parameterization, temporarily stopping the execution of the keystroke macro, allowing the user to type in the "parameter," and then finishing the execution.

Suppose a user has mailbox file which contains all the headers and text of messages received for a period of time and would like to create a shorter version of this file for quick reference containing only the headers. A macro to do this might look like this:

```
DEFINE MACRO mail_to_headers
deluntil /From /
skip 2
insert ------------------------------------------
next
DEFINE MACRO END
```

In this hypothetical editor, the user defines a macro called *mail_to_headers* by bracketing typical editor statements between a *DEFINE MACRO* and a *DEFINE MACRO END* bracket. The statements in this macro indicate to delete everything until the first line with the characters "From ", to skip 2 more lines (the "To" line and the "Subject" line in the mail message), to insert a line of underscores, and to go to the line after these new underscores.

Given, with the current line pointer pointing to a line preceeding the line "From wcs...":

```
From wcs Thu Mar  5 02:49:41 1981
To: nkm
Subject: (un)natural language processing
Cc: wcs

hi nrmn.

ths is a tst of th nw unix* cmnd avd. it mr  or is
bindy strps vwls frm stndd inpt & plcs th rslt
on stndd otpt.
  .
  .
  .

From avd Fri Mar  6 20:17:16 1981
To: nkm
Subject: Another lost cause
Cc: skf wp

Do you know where the excess fress resource manuals are, or the master
for that matter? Do you know how to copy it off the C-disk so someone
could take a look at it? Where the source is kept these days?

From skf Fri Mar  6 18:47:14 1981
To: fac grad graphics ugrad
Subject: The ACM Lecture Series: Judson Rosebush

On Thu Mar 12 @ 4pm in Pembroke Hall 210, the ACM will be sponsoring a
lecture by Judson Rosebush, president of Digital Effects, a NY-based
company doing commercial computer animation.
will be shown on film/videotape. The talk should be
```

successive invocations of the macro *mail_to_headers* would result in the file being changed to:

```
From wcs Thu Mar  5 02:49:41 1981
To: nkm
Subject: (un)natural language processing
------------------------------------------
From avd Fri Mar  6 20:17:16 1981
To: nkm
Subject: Another lost cause
------------------------------------------
From skf Fri Mar  6 18:47:14 1981
To: fac grad graphics ugrad
Subject: The ACM Lecture Series: Judson Rosebush
------------------------------------------
```

Figure 9. Example of macro facility.

3.4.3.3 Extensibility. Some editors allow the user to extend the command set, in the same language in which the editor is written. Thus the user is not limited to designing macros made up of editor primitives, but can design operations using the same lowest-level primitives as the nucleus of the editor uses itself. The fact that the extension is being done in an actual programming language, rather than a special-purpose editor macro language, implies greater efficiency and ease of expression of new functions. In EMACS, for instance, the editor can be used to modify or to create a function, and this function can be "linked" into the editor without ever leaving the editing environment. Of course, such a feature is not targeted to the general public but to more advanced users or programmers who are willing to learn the internals of the editor to modify or to add code.

3.4.4 Target-Specific Operations

Target-specific operations are those not common to all editors but specific to the target application area for which the editor is designed. A LISP program editor, for example, might have routines to locate matching parentheses. The target-specific operations are the most marked distinguishing characteristics among editors. These operations go further than manipulating elements simply as neutral parts of some larger whole; rather, they operate on these objects with some "knowledge" of what they are, for example, a capitalize command would need to know that a capital a was represented as an A. In syntax-directed program editors, for example, compilation of a syntactic entity is a target-specific operation. A more thorough description of these will be found in Part II in the discussion of actual editors.

4. CONCLUSION: PART I

While most editors, regardless of implementation and configuration, offer a set of frequently used, "standard" editing, traveling, viewing, and display functions, there is a wide difference in the number of more specialized features, as well as in the types of user interface available. This diversity is illustrated by the collection of editors described in "Interactive Editing Systems: Part II," which follows.